Technical Report
ES-XXX/05

Evolutive Maintenance: Observing Object-Oriented Software Decay

Marco Antonio Pereira Araújo
Guilherme Horta Travassos
Systems Engineering and Computer Science Program
Federal University of Rio de Janeiro - Brazil
COPPE/UFRJ
http://www.cos.ufrj.br

Barbara Kitchenham
Dept Computer Science
Keele University, UK
http://www.keele.ac.uk/depts/cs/

March/2005

1. Introduction

While the maintenance refers to the activities that happen in any time after the implementation of a new software development project, the software evolution is defined by the exam of the systems characteristics dynamic behavior and how they change along the time.

The Laws of Software Evolution (LSE) describe how a system behaves throughout their successive versions (LEHMAN, 1980). Works found in the literature makes reference to software evolution experimental studies just considering the legacy systems' source code (KEMERER & SLAUGHTER, 1999) (SCACHI, 2003). Besides, LEHMAN & RAMIL (2002) have been pointing out the need for evolution studies regarding object-oriented systems and in other software development process phases (LEHMAN & RAMIL, 2003). Due to these characteristics, decay causes' study throughout object-oriented development processes becomes relevant, providing us a better understanding of how this type of software evolves.

ARAUJO & TRAVASSOS (2004) have described the theme Software Evolution in the context of the Experimental Software Engineering, highlighting the concepts and characteristics involved in the area, as well as a bibliographical revision on the subject, with emphasis in experimental studies. The Laws of Software Evolution were used as the basis for a software evolution theory, originally proposed by LEHMAN (1980). In agreement with SCACHI (2003), this theory represents one of the largest intellectual contributions and challenges for the software evolution research community. KEMERER & SLAUGHTER (1999) state that only 2% of the experimental studies focus in maintenance, despites the fact that publications show at least 50% of the software effort is dedicated to this phase. Among the publications found in the literature, most refers to observational studies, usually concerned with the evolution in legacy systems' source code, some still regarding batch systems and, usually, in the COBOL language. The aim was to introduce the research that has been accomplished at COPPE/UFRJ for defining a framework, based on the Laws of Software Evolution (LEHMAN, 1980), for supporting experimental studies and decision making processes regarding object-oriented software decay. The purpose is the elaboration of a conceptual structure to support the definition of experimental studies for different object-oriented software development process phases.

In this work, we have evolved the scenario previously described (ARAUJO & TRAVASSOS, 2004), bringing the concepts concerned with the Laws of Software Evolution to the evolutive maintenance arena. Our aim is to explore the relationships among the LSE applying them to object-oriented software. With this, software engineers could be able to observe, by applying such model, the possible decay causes in their OO software projects. Besides, we believe this model can represent the first steps towards the building of a system dynamic model, what could support *in-virtuo* and *in-silico* experimental studies regarding evolutive maintenance and OO software decay

This technical report is divided in 2 more sections besides this introduction. The section 2 describes an initial discussion about the application of the Laws of Software Evolution in object-oriented software development processes, describing a framework for evolution studies in this context. Section 3 describes the final considerations and perspectives of future works.

2. Interpreting the Laws of Software Evolution when applied to Object-Oriented Software Development Processes

Once the approaches found in the technical literature usually treat of software evolution regarding legacy systems' source code (some software still in batch and, usually, written in the COBOL language), this work brings, as one of its intended contributions, the hypothesis that the Laws of Software Evolution (LSE) could also be supported by the different phases of a software development process based on the object-oriented paradigm, instead of just to the legacy systems' code phase. For this, without generality loss, an adaptation of the object-oriented software development process proposed in TRAVASSOS et al. (2001) was used, shown in the Figure 2.1.

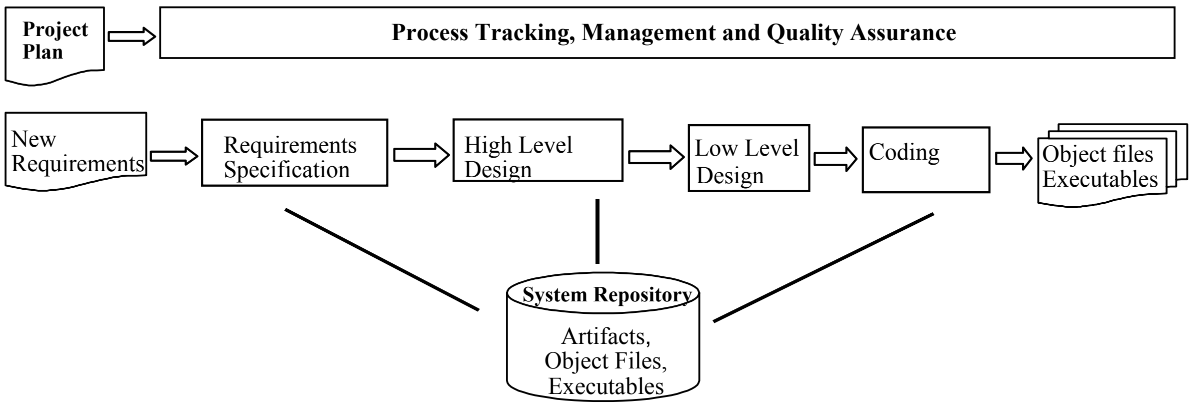
Project Plan
Process Tracking, Management and Quality Assurance
New Requirements
Requirements Specification
High Level Design
Low Level Design
Coding
Object files Executables
System Repository
Artifacts, Object Files, Executables

Figure 2.1 – The considered software development process (adapted from TRAVASSOS et al. (2001))

Considering this development scenario, firstly a set of characteristics that, when grouped, would affect OO software decay was identified, in order to provide a better understanding than it would really affect the software evolution. Some characteristics were adapted from ISO 9126-1 (1997) and others have been added in the sense of contemplating characteristics not included by this standard, but relevant in the software evolution process.

Among the extracted and adapted characteristics from ISO 9126-1, there are Reliability, Efficiency and Maintainability. The added characteristics are Size, Periodicity, Complexity, Effort and Modularity. These characteristics are measured through the collection of specific metrics to each artifact version.

We characterize Size as the amount of artifacts produced in each phase of the proposed software development process, as amount of function points for Requirements Specification, amount of key classes for High Level Design, amount of support classes for Low Level Design and number of source code lines for Coding.

As Periodicity we are representing the interval of time elapsed between each produced version of that artifact.

Complexity, in the context of this work, is related only to Structural Complexity, which is measurable, and will be named just Complexity throughout this text. Thus, Complexity is identified through elements that can measure the structural complexity of the artifact, as number of requirements in a requirements document, number of class diagrams and cyclomatic complexity per method, exemplifying, respectively, artifacts of Requirements Specification phases, High Level Design and Coding.

As Effort we considered as the amount of artifacts handled (number of inclusions, modifications and exclusions in each artifact). Besides, in case of modifications, an artifact modified several times is counted repeatedly as many times as the modifications are made.

We describe Modularity through the coupling and cohesion characteristics between artifacts as, for instance, coupling between Use Cases in Requirements Specification, between classes in High and Low Level Design and cohesion in methods in Coding.

As Reliability we represent the amount of identified defects by artifact in each version of it, besides system's availability. This characteristic was based mainly on the Maturity sub-characteristic from ISO 9126-1.

Efficiency is identified by the amount of people and allocated resources, spent time and average productivity of the team, by version of each artifact. This characteristic was based mainly on the Time Behavior and Resource Behavior sub-characteristics from ISO 9126-1.

Finally, Maintainability is characterized by the spent time in the identification of defects and also for the spent time in their removal. This characteristic was mainly based on the Changeability and Testability sub-characteristics from ISO 9126-1.

The table 2.1 summarizes the relationships among the Laws of Software Evolution and the described characteristics.

Table 2.1 – Laws of Software Evolution x Characteristics

	Size	Periodicity	Complexity	Effort	Modularity	Reliability	Efficiency	Maintainability
Continuing Change		✓		✓				
Increasing Complexity	✓		✓	✓	✓			✓
Self Regulation	✓					✓	✓	
Conservation of Organizational Stability				✓			✓	
Conservation of Familiarity	✓		✓	✓				
Continuing Growth	✓	✓						
Declining Quality			✓	✓	✓	✓		✓
Feedback System	✓	✓	✓	✓	✓	✓	✓	✓

Starting from the definition of the characteristics that influence in the software evolution, a study was made in how these characteristics would behave to result in the software decay. These characteristics were grouped and interpreted according to the perspective given by the LSE and OO software development phase. It aimed at capturing each expected characteristic's trend when evaluating software decay. We used truth tables to capture each one of the possible combinations of characteristics' trends. Next, based on the combination of these trends, we marked down those ones that are applicable to the LSE. At the end, the combination of these trends gives the hypotheses regarding the LSE and associated characteristics' trends. After each table, there comes the associated study hypothesis, described through the formalism defined by CARVER (2003). The following symbolism will be used for the tables construction:

- $\uparrow$ increasing of a specific characteristic;
- $\downarrow$ decreasing of a specific characteristic;
- $\leftrightarrow$ a characteristic is constant, that is, doesn't change;
- * regardless of its value;
- $\times$ lack of applicability to the Law of Software Evolution;
- ✓ applicability to the Law of Software Evolution;
- $\wedge$ connective AND;
- $\vee$ connective OR;
- $\neg$ connective NOT;
- $\Rightarrow$ logical imply.

The Law of Continuing Change (Law I)

Original interpretation: “An E-type[1] system must be continually adapted else it becomes progressively less satisfactory in use” (LEHMAN & RAMIL, 2001b). The need for change reflects a need to adapt the system as the outside world, the domain being covered and the application and/or activity being supported or pursued, changes. Such exogenous changes are likely to invalidate assumptions made during system definition, development, validation, installation and application or render them unsatisfactory. The software reflecting such assumptions must then be adapted to restore their validity (LEHMAN & RAMIL, 2001b).

In this work, the Law of Continuing Change (CC) is being characterized by the number of accomplished modifications (additions, removals, modifications) over the successive versions of the system artifacts, taking into consideration the time interval between versions. A system is considered to be undergoing continuing change if new versions are produced at regular time intervals, showing then modifications in functionality and/or structure.

The table 2.2 shows the expected impact of each defined characteristic to observe the Law of Continuing Change, describing, at the end, the logical formulation regarding the software characteristics behavior and the Law I.

Table 2.2 – Truth Table for the Law of Continuing Change

Periodicity	Effort	Continuing Change
↑	*	×
↔	↑	✓
↔	↔	✓
↔	↓	×
↓	↑	✓
↓	↔	✓
↓	↓	×
¬↑ ∧	**¬↓ ⇒**	**CC**

This way, we described the following hypothesis to observe this law:

(Periodicity doesn’t increase ∧ Effort doesn’t decrease) ⇒ Continuing Change

The Law of Increasing Complexity (Law II)

Original interpretation: “As an E-type system is evolved its complexity increases unless work is done to maintain or reduce it” (LEHMAN & RAMIL, 2001b). Increasing complexity arises because of the injection and the super-positioning of changes to achieve, for example, growth in functionality of satisfaction of the needs of changing operational domains. This leads to increasing internal interconnectivity and, hence, to deteriorating system structure, increasing disorder. Equally, it results in increasing complexity of internal and external interfaces at all levels. These effects are amplified because, as the system ages, changes are more likely to be orthogonal to existing system structures (LEHMAN & RAMIL, 2001b).

[1] An *E*-type system represents software solving a problem or addressing an application in the real word.

In this context, the Law of Increasing Complexity (IC) is being characterized by the increase of the characteristics that mainly lead to the system's structural complexity deterioration, thus leading to an approach which aims at the reduction and/or impairment of such process. This increase in complexity is observed by means of the number of produced artifacts, structural complexity, number of modifications already done, loss of artifacts modularity (through coupling increases or cohesion loss) as well as the system maintainability decrease.

The table 2.3 shows the expected impact of each defined characteristic to observe the Law of Increasing Complexity, describing, at the end, the logical formulation regarding the software characteristics behavior and the Law II.

Table 2.3 – Truth Table for the Law of Increasing Complexity

Size	Complexity	Effort	Modularity	Maintainability	Increasing Complexity
↑	*	*	*	*	✓
↔	↑	*	*	*	✓
↔	↔	↑	*	*	✓
↔	↔	↔	↑	↑	×
↔	↔	↔	↑	↔	×
↔	↔	↔	↑	↓	✓
↔	↔	↔	↔	↑	×
↔	↔	↔	↔	↔	×
↔	↔	↔	↔	↓	✓
↔	↔	↔	↓	*	✓
↔	↔	↓	↑	↑	×
↔	↔	↓	↑	↔	×
↔	↔	↓	↑	↓	✓
↔	↔	↓	↔	↑	×
↔	↔	↓	↔	↔	×
↔	↔	↓	↔	↓	✓
↔	↔	↓	↓	*	✓
↔	↓	↑	*	*	✓
↔	↓	↔	↑	↑	×
↔	↓	↔	↑	↔	×
↔	↓	↔	↑	↓	✓
↔	↓	↔	↔	↑	×
↔	↓	↔	↔	↔	×
↔	↓	↔	↔	↓	✓
↔	↓	↔	↓	*	✓
↔	↓	↓	↑	↑	×
↔	↓	↓	↑	↔	×
↔	↓	↓	↑	↓	✓
↔	↓	↓	↔	↑	×
↔	↓	↓	↔	↔	×
↔	↓	↓	↔	↓	✓
↔	↓	↓	↓	*	✓
↓	↑	*	*	*	✓
↓	↔	↑	*	*	✓
↓	↔	↔	↑	↑	×
↓	↔	↔	↑	↔	×
↓	↔	↔	↑	↓	✓
↓	↔	↔	↔	↑	×
↓	↔	↔	↔	↔	×
↓	↔	↔	↔	↓	✓
↓	↔	↔	↓	*	✓
↓	↔	↓	↑	↑	×
↓	↔	↓	↑	↔	×
↓	↔	↓	↑	↓	✓
↓	↔	↓	↔	↑	×
↓	↔	↓	↔	↔	×
↓	↔	↓	↔	↓	✓
↓	↔	↓	↓	*	✓
↓	↓	↑	*	*	✓
↓	↓	↔	↑	↑	×

Size	Complexity	Effort	Modularity	Maintainability	Increasing Complexity
↓	↓	↔	↑	↔	×
↓	↓	↔	↑	↓	✓
↓	↓	↔	↔	↑	×
↓	↓	↔	↔	↔	×
↓	↓	↔	↔	↓	✓
↓	↓	↔	↓	*	✓
↓	↓	↓	↑	↑	×
↓	↓	↓	↑	↔	×
↓	↓	↓	↑	↓	✓
↓	↓	↓	↔	↑	×
↓	↓	↓	↔	↔	×
↓	↓	↓	↔	↓	✓
↓	↓	↓	↓	*	✓
↑ v	↑ v	↑ v	↓ v	↓ ⇒	**IC**

This way, we described the following hypothesis to observe this law:
(Size increases **V** Complexity increases **V** Effort increases **V** Modularity decreases **V** Maintainability decreases) ⇒ Increasing Complexity

The Law of Self Regulation (Law III)

Original interpretation: “Global E-type system evolution processes are self regulating” (LEHMAN & RAMIL, 2001b). The global process includes all activities that influence the software process and its product, and includes not only direct technical activity but also that of other stakeholders such as business executives, marketers, users and their managers (LEHMAN & RAMIL, 2000).

In the context of this work, the Law of Self Regulation (SR) is being characterized by keeping the system development under control by means of measuring the system’s reliability, allocated resources throughout successive versions of maintenance with increases in the system size. Reliability can be measured through the number of detected and corrected defects for each system version as well its availability ratio. Allocated resources can be measured based on the team’s size and productivity, allocated time and consumed resources.

The table 2.4 shows the expected impact of each defined characteristic to study the Law of Self Regulation, describing, at the end, the logical formulation concerned with the software characteristics behavior and Law III.

Table 2.4 – Truth Table for the Law of Self Regulation

Size	Reliability	Efficiency	Self Regulation
↑	*	*	×
↔	↑	↑	✓
↔	↑	↔	✓
↔	↑	↓	×
↔	↔	↑	✓
↔	↔	↔	✓
↔	↔	↓	×
↔	↓	*	×
↓	↑	↑	✓
↓	↑	↔	✓
↓	↑	↓	×
↓	↔	↑	✓
↓	↔	↔	✓
↓	↔	↓	×
↓	↓	*	×
¬↑ ^	¬↓ ^	¬↓ ⇒	**SR**

This way, we described the following hypothesis to observe this law:
(Size doesn’t increase **Λ** Reliability doesn’t decrease **Λ** Efficiency doesn’t decrease) ⇒ Self Regulation

The Law of Conservation of Organizational Stability (Law IV)

Original interpretation: “Average activity rate in an E-type process tends to remain constant over system lifetime or segments of that lifetime” (LEHMAN & RAMIL, 2001b). The activity rate (e. g., elements changed, handled or handlings per release or unit of time tends to remain constant over periods or segments of system lifetime (LEHMAN & RAMIL, 2001b)

In this work, the Law of Conservation of Organizational Stability (COS) is being characterized by the stagnation of the number of elements changed and allocated resources, showing that the added functionality, or the increase of allocated resources, don’t significantly change it. It can be checked by means of the number of changes’ constancy and resources allocated to it.

The table 2.5 shows the expected impact of each one of the software characteristics to observe the Law of Conservation of Organizational Stability, describing, at the end, the logical formulation regarding software characteristics behavior and Law IV.

Table 2.5 – Truth Table for the Law of Conservation of Organizational Stability

Effort	Efficiency	Conservation of Organizational Stability
↑	*	×
↔	↑	×
↔	↔	✓
↔	↓	×
↓	*	×
↔ ∧	↔ ⇒	**COS**

This way, we described the following hypothesis to observe this law:
(Effort doesn’t change **Λ** Efficiency doesn’t change) ⇒ Conservation of Organizational Stability

The Law of Conservation of Familiarity (Law V)

Original interpretation: “In general, the average incremental growth (growth rate trend) of E-type systems tends to decline” (LEHMAN & RAMIL, 2001b). Give the growing complexity of the system, its workings and its functionality, achieving renewed familiarity after numerous changes, additions and removals, restoration of pre-change familiarity after change becomes increasingly difficult. This reasoning suggests that the rate of change and growth of the system be slowed down as it ages (LEHMAN & RAMIL, 2001b).

In this work, the Law of Conservation of Familiarity (CF) is being characterized by the point from which all the modifications added into the system’s versions show little difference in general functionality. It can be observed by the constancy in system size, its complexity and number of manipulations (additions, removals, modifications) on the artifacts of the system.

The table 2.6 shows the expected impact of each defined software characteristic to observe the Law of Conservation of Familiarity, describing, at the end, the logical formulation concerned with the characteristics behavior and Law V.

Table 2.6 – Truth Table for the Law of Conservation of Familiarity

Size	Complexity	Effort	Conservation of Familiarity
↑	*	*	×
↔	↑	*	×
↔	↔	↑	×
↔	↔	↔	✓
↔	↔	↓	×
↔	↓	*	×
↓	*	*	×
↔ ∧	↔ ∧	↔ ⇒	**CF**

This way, we described the following hypothesis to observe this law:

(Size doesn't change Λ Complexity doesn't change Λ Effort doesn't change) ⇒ Conservation of Familiarity

The Law of Continuing Growth (Law VI)

Original interpretation: "The functional capability of E-type systems must be continually increased to maintain user satisfaction over the system lifetime" (LEHMAN & RAMIL, 2001b). This law is more related to the fact that finiteness of the implemented software implies that its properties are bounded relative to those of the application and its domain. Properties excluded by the bounds eventually become a source of performance limitations, irritation and error. To eliminate the latter requires extension of the system (LEHMAN & RAMIL, 2000).

In this context, the Law of Continuing Growth (CG) is being characterized by the continuing increase of functionality offered by the system. It's noticeable by the quantity of existing artifacts for each new version that should be released at regular intervals.

The table 2.7 shows the expected impact of each defined characteristic to observe the Law of Continuing Growth, describing, at the end, the logical formulation regarding the characteristics behavior and Law VI.

Table 2.7 – Truth Table for the Law of Continuing Growth

Size	Periodicity	Continuing Growth
↑	↑	×
↑	↔	✓
↑	↓	✓
↔	*	×
↓	*	×
↑ ∧	¬↑ ⇒	**CG**

This way, we described the following hypothesis to observe this law:

(Size increases Λ Periodicity doesn't increase) ⇒ Continuing Growth

The Law of Declining Quality (Law VII)

Original interpretation: "Unless rigorously adapted to take into account changes in the operational environment, the quality of an E-type systems will appear to decline as it is evolved" (LEHMAN & RAMIL, 2001b). Assuming quality to be related to the extend which system behavior address the needs of an application and its domain, the law derives from the observation that user needs inevitably change with time (LEHMAN & RAMIL, 2000).

In the context of this work, the Law of Declining Quality (DQ) expresses the loss of quality of a given system, leading to improvements in order to avoid decay. It can be measured through the increase in structural complexity, increase in the number of manipulations (additions, removals, modifications) over the artifacts, modularity loss, maintainability and reliability decrease.

The table 2.8 shows the expected impact of each defined characteristic to observe the Law of Declining Quality, describing, at the end, the logical formulation concerned with the software characteristics behavior and Law VII.

Table 2.8 – Truth Table for the Law of Declining Quality

Complexity	Effort	Modularity	Reliability	Maintainability	Declining Quality
↑	*	*	*	*	✓
↔	↑	*	*	*	✓
↔	↔	↑	↑	↑	×
↔	↔	↑	↑	↔	×
↔	↔	↑	↑	↓	✓
↔	↔	↑	↔	↑	×
↔	↔	↑	↔	↔	×
↔	↔	↑	↔	↓	✓
↔	↔	↑	↓	*	✓
↔	↔	↔	↑	↑	×
↔	↔	↔	↑	↔	×
↔	↔	↔	↑	↓	✓
↔	↔	↔	↔	↑	×
↔	↔	↔	↔	↔	×
↔	↔	↔	↔	↓	✓
↔	↔	↔	↓	*	✓
↔	↔	↓	*	*	✓
↔	↓	↑	↑	↑	×
↔	↓	↑	↑	↔	×
↔	↓	↑	↑	↓	✓
↔	↓	↑	↔	↑	×
↔	↓	↑	↔	↔	×
↔	↓	↑	↔	↓	✓
↔	↓	↑	↓	*	✓
↔	↓	↔	↑	↑	×
↔	↓	↔	↑	↔	×
↔	↓	↔	↑	↓	✓
↔	↓	↔	↔	↑	×
↔	↓	↔	↔	↔	×
↔	↓	↔	↔	↓	✓
↔	↓	↔	↓	*	✓
↔	↓	↓	*	*	✓
↓	↑	*	*	*	✓
↓	↔	↑	↑	↑	×
↓	↔	↑	↑	↔	×
↓	↔	↑	↑	↓	✓
↓	↔	↑	↔	↑	×
↓	↔	↑	↔	↔	×
↓	↔	↑	↔	↓	✓
↓	↔	↑	↓	*	✓
↓	↔	↔	↑	↑	×
↓	↔	↔	↑	↔	×
↓	↔	↔	↑	↓	✓
↓	↔	↔	↔	↑	×
↓	↔	↔	↔	↔	×
↓	↔	↔	↔	↓	✓
↓	↔	↔	↓	*	✓
↓	↔	↓	*	*	✓

Complexity	Effort	Modularity	Reliability	Maintainability	Declining Quality
↓	↓	↑	↑	↑	×
↓	↓	↑	↑	↔	×
↓	↓	↑	↑	↓	✓
↓	↓	↑	↔	↑	×
↓	↓	↑	↔	↔	×
↓	↓	↑	↔	↓	✓
↓	↓	↑	↓	*	✓
↓	↓	↔	↑	↑	×
↓	↓	↔	↑	↔	×
↓	↓	↔	↑	↓	✓
↓	↓	↔	↔	↑	×
↓	↓	↔	↔	↔	×
↓	↓	↔	↔	↓	✓
↓	↓	↔	↓	*	✓
↓	↓	↓	*	*	✓
↑ **v**	↑ **v**	↓ **v**	↓ **v**	↓ ⇒	**DQ**

This way, we described the following hypothesis to observe of this law:
(Complexity increases **V** Effort increases **V** Modularity decreases **V** Reliability decreases **V** Maintainability decreases) ⇒ Declining Quality

The Law of Feedback System (Law VIII)

Original interpretation: “E-type evolution processes are multi-level, multi-loop, multi-agent feedback systems” (LEHMAN & RAMIL, 2001b). The behavior of complex feedback systems is not and cannot, in general, be described directly in terms of the local behavior of its forward path activities and mechanisms. Feedback will constrain the ways that the process constituents interact with one another and will modify their individual, local, and collective, global, behavior (LEHMAN & RAMIL, 2001b).

In this work, the Law of Feedback System (FS) is characterized by the collecting and analyzing of all prior LSE, given us information not only on the understanding of the evolution process itself but also on the decision-making process regarding the application of corrective and preventive means related to system decay. We interpreted this law as a result of the analyses of the previous laws and, therefore, it doesn't describe a specific true table for its logical definition.

This way, we described the following hypothesis to observe this law:
(Collection of relative measures to Size, Periodicity, Complexity, Modularity, Effort, Reliability, Efficiency, Maintainability) ⇒ Feedback System

The Hypotheses for the Laws of Software Evolution

Using these truth tables, it was possible to describe a set of hypotheses, each one associated to a specific LSE. The table 2.9 summarizes these hypotheses.

Table 2.9 – The Hypotheses for the Laws of Software Evolution

Hypotheses
(Periodicity doesn’t increase **Λ** Effort doesn’t decrease) ⇒ Continuing Change
(Size increases **V** Complexity increases **V** Effort increases **V** Modularity decreases **V** Maintainability decreases) ⇒ Increasing Complexity
(Size doesn’t increase **Λ** Reliability doesn’t decrease **Λ** Efficiency doesn’t decrease) ⇒ Self Regulation
(Effort doesn’t change **Λ** Efficiency doesn’t change) ⇒ Conservation of Organizational Stability
(Size doesn’t change **Λ** Complexity doesn’t change **Λ** Effort doesn’t change) ⇒ Conservation of Familiarity

(Size increases Λ Periodicity doesn't increase) ⇒ Continuing Growth
(Complexity increases **V** Effort increases **V** Modularity decreases **V** Reliability decreases **V** Maintainability decreases) ⇒ Declining Quality
(Collection of relative measures to Size, Periodicity, Complexity, Effort, Modularity, Reliability, Efficiency, Maintainability) ⇒ Feedback System

Linkages among the Laws of Software Evolution

Kitchenham (1982) and Lehman & Ramil (2001b) show that the Laws of Software Evolution are not independent. Thus, the presented hypotheses were discussed in order to identify the linkages among the Laws of Software Evolution (figure 2.2). This discussion is particularly relevant because this work is concerned with identifying software decay process and not to study the individual behavior of the Laws of Software Evolution.

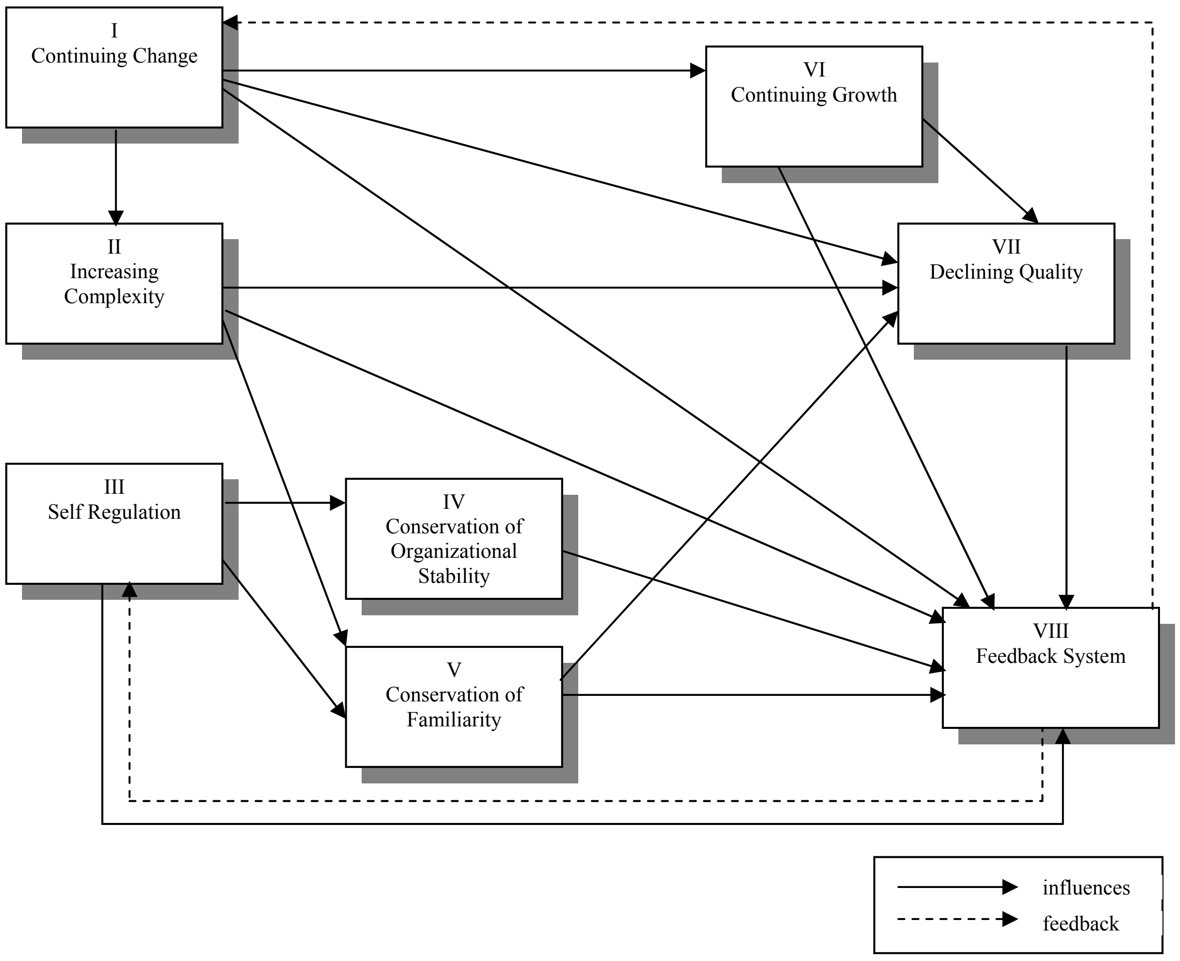

Figure 2.2 – Linkages among the Laws of Software Evolution

However, it's important to describe how the Laws of Software Evolution influence each other. Thus, figure 2.4 shows these linkages by using the SADT (Structured Analysis and Design Technique) diagram (ROSS, 1977), considering the presented hypotheses logical formulation. The structure of a SADT diagram is represented by figure 2.3, where Input are the received data by means of an activity, Output are the generated or transformed data by an activity, Control are data whose utilization influence on the input/output transformation process and Mechanism is the processor that undertakes or processes the activity.

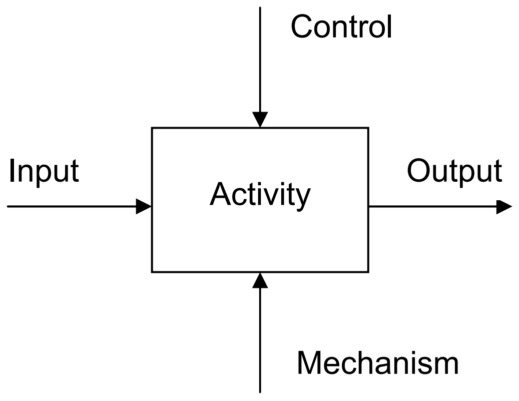

Figure 2.3 – Representation of a SADT diagram

The established hypotheses are in our view the definition bases of the Laws of Software Evolution causes. For example, the Law of Conservation of Familiarity (Law V) has as input the system's current version. For this Law, the controls are represented by the system's baseline and the software characteristics represented by Size, Complexity and Effort. Also as controls, we have the status of Laws II and III that indicate the possibility of these Laws be influencing over the current Law (Law V). This fact indicates that Laws II and III have straight influence in Law V; that is, if the presented hypotheses are true, we should also verify their impact in the laws that could be directly affected, such as the Law of Conservation of Familiarity in this example. The mechanism is represented by the logical formulation for this Law, as described in table 2.9. The output is the status of Law V, in addition to the fact that this Law may influence the Law of Declining Quality (Law VII) and Law of Feedback System (Law VIII).

However, we can't claim that a specific Law of Software Evolution, which its associated logical proposition is true, results in software decay. This is particularly relevant when considering anti-regressive work (LEHMAN & RAMIL, 2001) (PFLEEGER, 2001) in relation to decay, like redocumentation, restructuring and reverse engineering. Thus, Table 2.10 shows a discussion of each Law of Software Evolution regarding its neutral feature and its positive and negative implications. It's relevant to remark that the neutral feature reveals an inherent characteristic towards the Law of Software Evolution itself. The positive implication reveals situations where the Law might result in an opposite condition to the software decay, like the use of anti-regressive work, for example. The negative implication is the one that results in decay and describes the interests in this work. Thus, we start to interpret the presented hypotheses through the negative implication for each Law of Software Evolution.

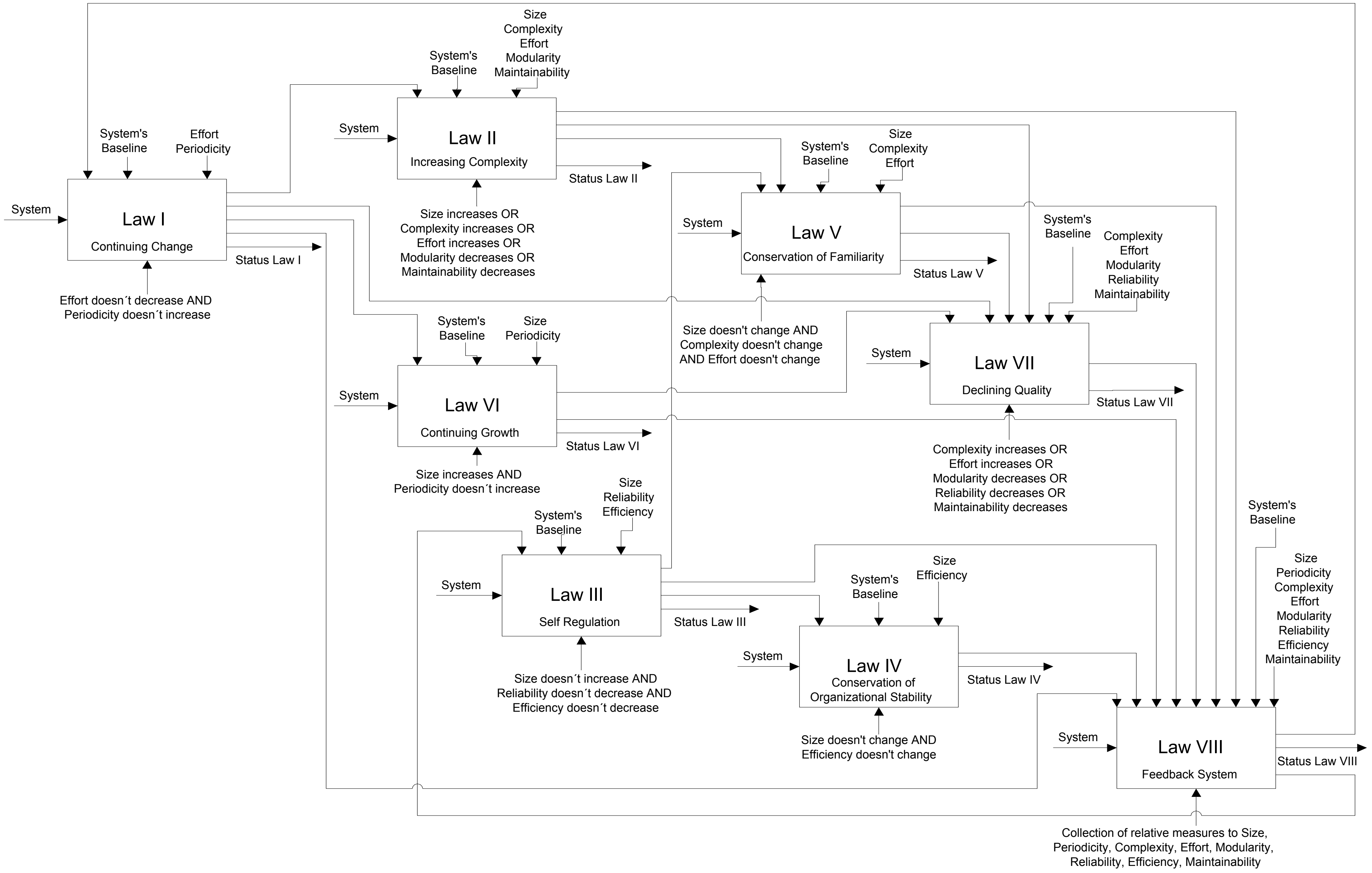
Law I
Continuing Change
System
System's Baseline
Effort Periodicity
Status Law I
Effort doesn´t decrease AND Periodicity doesn´t increase
Law II
Increasing Complexity
System
System's Baseline
Size Complexity Effort Modularity Maintainability
Status Law II
Size increases OR Complexity increases OR Effort increases OR Modularity decreases OR Maintainability decreases
Law V
Conservation of Familiarity
System
System's Baseline
Size Complexity Effort
Status Law V
Size doesn't change AND Complexity doesn't change AND Effort doesn't change
Law VI
Continuing Growth
System
System's Baseline
Size Periodicity
Status Law VI
Size increases AND Periodicity doesn´t increase
Law VII
Declining Quality
System
System's Baseline
Complexity Effort Modularity Reliability Maintainability
Status Law VII
Complexity increases OR Effort increases OR Modularity decreases OR Reliability decreases OR Maintainability decreases
Law III
Self Regulation
System
System's Baseline
Size Reliability Efficiency
Status Law III
Size doesn´t increase AND Reliability doesn´t decrease AND Efficiency doesn´t decrease
Law IV
Conservation of Organizational Stability
System
System's Baseline
Size Efficiency
Status Law IV
Size doesn't change AND Efficiency doesn't change
Law VIII
Feedback System
System
System's Baseline
Size Periodicity Complexity Effort Modularity Reliability Efficiency Maintainability
Status Law VIII
Collection of relative measures to Size, Periodicity, Complexity, Effort, Modularity, Reliability, Efficiency, Maintainability

Figure 2.4 – Linkages among the Laws of Software Evolution by using a SADT Diagram

Table 2.10 – Neutral Feature, Positive and Negative Implications for the Laws of Software Evolution

Law I – Continuing Change: An *E*-type system must be continually adapted else it becomes progressively less satisfactory in use
Neutral: A product in use will be subject to continuous change **Positive Implication**: If the change is well performed, the product will remain useful reflecting the changes in the environment and in the users' expectations **Negative Implication**: If the change is badly performed or the product is not modified, the product will become progressively less useful
Law II – Increasing Complexity: As an *E*-type system is evolved its complexity increases unless work is done to maintain or reduce it
Neutral: A modified product will become more complex **Positive Implication**: Increasing in complexity might be necessary to reflect modifications in the product environment and the increasing of users' needs **Negative Implications**: Increasing in complexity might turn the product more difficult to maintain and use
Law III – Self Regulation: *Global E*-type system evolution processes are self-regulating
Neutral: Systems have self regulation mechanisms **Positive Implication**: Self regulation allows the management of the evolution systems' process, resulting in a balance that might be explored for its good management **Negative Implication**: Self regulation is not considered nor used in the systems' development management, turning into barriers for a good management
Law IV – Conservation of Organisational Stability: Average activity rate in an *E*-type process tends to remain constant over system lifetime or segments of that lifetime
Neutral: Systems have global activities rates along their lifetime **Positive Implication**: The global activities rates tend to remain constant over system lifetime, indicating possible maturation and stability in the system's development **Negative Implication**: The global activity rates over lifetime vary significantly, indicating an instability in the system's development. Negative variations indicate the possibility of discontinuing product, while positive ones indicate the possibility of excessive effort for changes
Law V – Conservation of Familiarity: In general, the average incremental growth (growth rate trend) of *E*-type systems tends to decline
Neutral: Growth rate trend of systems can be measured **Positive Implication**: Average incremental growth rates tend to decline, indicating stability in the system's development **Negative Implication**: Average incremental growth rates tend to increase or remain constant evidencing the possibility of excessive effort between system's evolution cycles, indicating instability in the system's development
Law VI – Continuing Growth: The functional capability of *E*-type systems must be continually increased to maintain user satisfaction over the system lifetime
Neutral: Systems in use are subject to continuing growth **Positive Implication**: The increment of new functionality can keep the system utility, remaining useful related to changes in the environment and the user's expectations **Negative Implication**: The lack of new functionality increment may turn the system progressively less useful according to the user's needs
Law VII – Declining Quality: Unless rigorously adapted to take into account changes in the operational environment, the quality of an *E*-type system will appear to decline as it is evolved
Neutral: A system in evolution are subject to quality decay **Positive Implication**: Might result in anti-regressive work, like redocumentation, restructuring, reengineering, reverse engineering and elimination of useless artifacts **Negative Implication**: Declining in quality may turn the system more difficult to understand and change
Law VIII – Feedback System: *E*-type evolution processes are multi-level, multi-loop, multi-agent *feedback* systems
Neutral: Systems in evolution have feedback mechanisms **Positive Implication**: The feedback system is used to reach relevant enhancement **Negative Implication**: The feedback system is not considered nor used in order to reach relevant enhancement

Based on these studies, we believe that these linkages are applicable not only inside each phase of the software development process, like Requirements Specifications, High and Low Level Design and Coding, but also through these phases in a software development process (figure 2.5). Therefore, we propose a hypothesis based on the fact that the software characteristics previously defined, which logically determines the Laws of Software Evolution behavior, should be investigated throughout the software development phases, considering the linkages among the Laws of Software Evolution, in order to identify the software decay causes. For example, an Increasing of Complexity in Requirements Specification should be investigated in High Level Project and thus, successively, in the next phases. This thought should also be valid for the other Laws of Software Evolution. This hypothesis needs some experimental investigation, which will be an object of study in this research's scope.

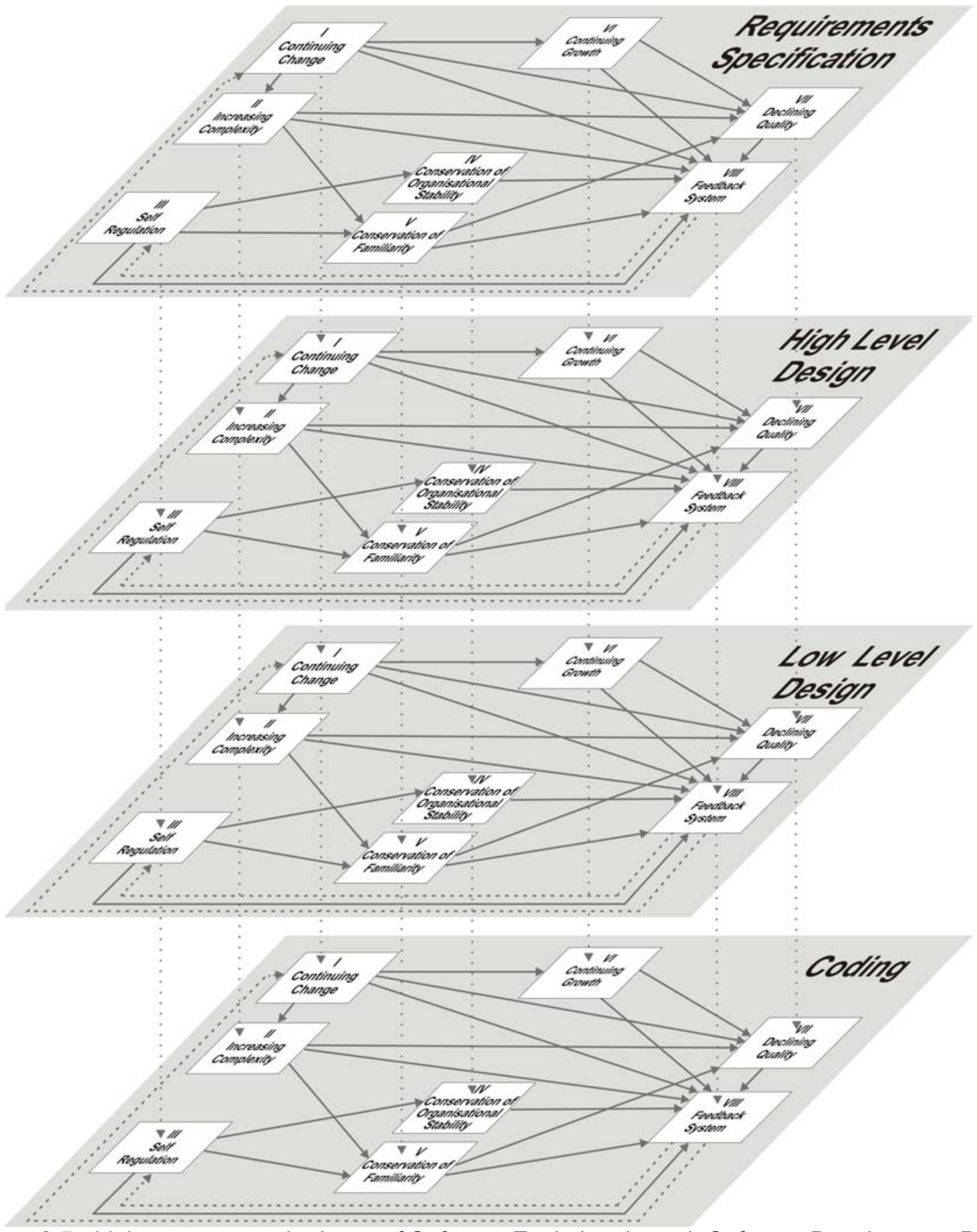

Figure 2.5 - Linkages among the Laws of Software Evolution through Software Development Phases

Besides, we believe that each version produced in this software evolution process influences next software versions decay. Regarding the fact that we are talking about evolutive maintenance, we established that a group of new requirements is associated to each new artifact version and, from them on, we should map the linkages among the Laws of Software Evolution not only among the development phases of one version (figure 2.5), but also capturing the impact of these modifications for the next software versions (figure 2.6). Thus, our start study point should be always concerned with the Requirements Specification phase, which is mapped in the next phases inside the referred maintenance cycle and, next, between the versions of that artifact. Needless to say that, despite the fact that the Requirements Specification is always the start point of any software project, it is possible to map the linkages among the development phases up to the abstraction level desired for the studies, that is, we could study the evolution process just up to High Level Project phase, for example. This brings out the discussion of software evolution for higher abstraction levels rather than mainly considering just the Coding phase. These hypotheses also need experimental investigation and also must be object of future studies in the context of this research.

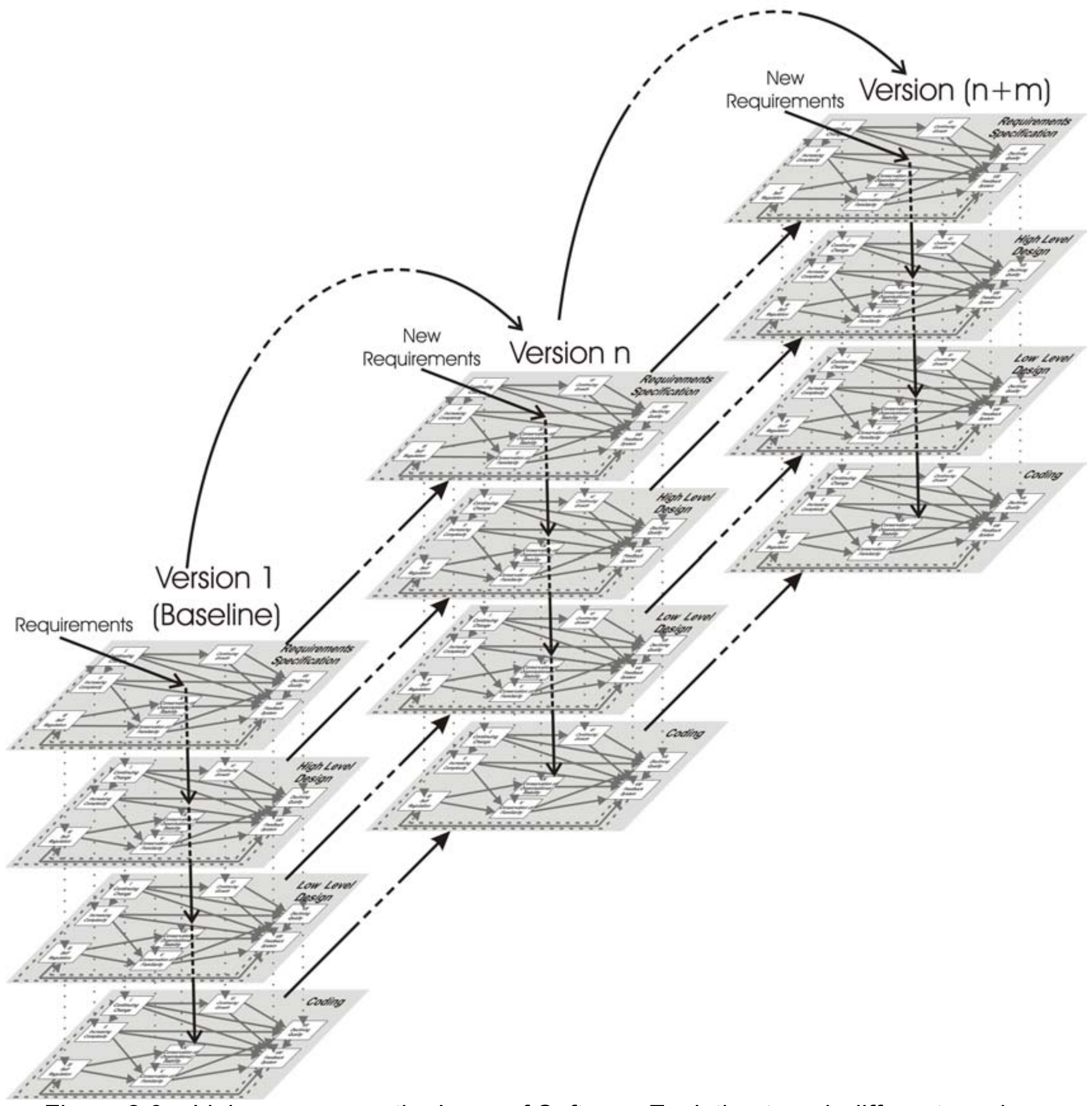

Figure 2.6 – Linkages among the Laws of Software Evolution trough different versions produced throughout the software development process

Mapping the hypotheses to the object-oriented software metrics

Starting from this set of hypotheses and, considering the software development process using the object-oriented paradigm previously proposed, we defined a set of metrics that would be applicable to each development process phase.

This set of metrics defines a parameter based framework, allowing flexibility to collect their values and to study the software decay inside one of the development process phases. This approach turns the study of the software evolution softer, once it can be adapted regarding the development process used, just collecting those metrics that could be extracted from the process in use. This flexibility will be more evident when associating the identified hypotheses with the used process phases and the defined metrics. This mapping will be done later in this work and it will indicate the relationship between the metrics and its optional use.

The metrics that give support to each one of the characteristics described previously in the different development process phases were based, generically, in the works of PFLEEGER (2001) and PRESSMAN (2001). More specific metrics related to the context of the object-oriented paradigm were based on CHIDAMBER & KEMERER (1994), LORENZ & KIDD (1994), TRAVASSOS et al. (2001) and TRAVASSOS (2003).

The set of metrics associated to each characteristic in each development process phase is represented in the table 2.11.

Table 2.11 - Metrics associated by Characteristic in each phase of the Process

	Size	Periodicity	Complexity	Effort	Modularity	Reliability	Efficiency	Maintainability
Requirements Specification	• # Function Points • # Use Case Points	• Interval between Versions	• # Requirements • # Use Cases	• # Requirements Handled • # Use Cases Handled	• Coupling between Use Cases (# Extensions and Uses)	• # Detected Defects • # Corrected Defects	• # People • Allocated Resources • Spent Time • Average Productivity of the Team	• Spent Time in the Diagnostic of Defects • Spent Time in the Removal of Defects
High Level Design	• # Classes • # Methods per Class	• Interval between Versions	• # Class Diagrams • # Sequence Diagrams • # State Diagrams • # Package Diagrams • # Activity Diagrams • Depth of Inheritance per Class • # Children per Class	• # Class Diagrams Handled • # Sequence Diagrams Handled • # State Diagrams Handled • # Package Diagrams Handled • # Activity Diagrams Handled	• Coupling between Classes	• # Detected Defects • # Corrected Defects	• # People • Allocated Resources • Spent Time • Average Productivity of the Team	• Spent Time in the Diagnostic of Defects • Spent Time in the Removal of Defects
Low Level Design	• # Key Classes • # Support Classes • # Methods per Class • # Subsystems	• Interval between Versions	• # Class Diagrams • # Sequence Diagrams • Depth of Inheritance per Class • Coupling between Objects • Response for a Class • Lack of Cohesion in Methods • # Children per Class	• # Class Diagrams Handled • # Sequence Diagrams Handled	• Cohesion in Methods • Coupling between Classes	• # Detected Defects • # Corrected Defects	• # People • Allocated Resources • Spent Time • Average Productivity of the Team	• Spent Time in the Diagnostic of Defects • Spent Time in the Removal of Defects
Coding	• # Lines of Source Code • # Methods per Class	• Interval between Versions	• Depth of Inheritance per Class • Coupling between Objects • Response for a Class • Lack of Cohesion in Methods • # Children per Class • Cyclomatic Complexity per Method	• # Lines of Source Code Handled	• Cohesion in Methods • Coupling between Classes	• # Detected Defects • # Corrected Defects • System Availability	• # People • Allocated Resources • Spent Time • Average Productivity of the Team	• Spent Time in the Diagnostic of Defects • Spent Time in the Removal of Defects

Based on this set of metrics, a study could be made in order to verify the trend of these metrics for the hypotheses previously described.

For each software development process phase, a table was elaborated relating the metrics identified in the table 2.11 through the interpretation of the described hypotheses, in other words, relating metrics with the Laws of Software Evolution, Characteristics and Relationships between the Characteristics, for each phase of the process. In this context, the table 2.12 shows the interpretation of the metrics in Requirements Specification phase by Characteristic and Law of Software Evolution, while the tables 2.13, 2.14 and 2.15 describe the interpretations of the metrics for the High Level Design, Low Level Design and Code phases, respectively.

These tables show, among the columns that identify the Characteristics, logical connectives that map the hypotheses. Inside of each cell of the tables, the metrics are described, with their respective interpretations related with the hypotheses. Logical connectives among these metrics indicate the obligation or not of the collection. The logical connective ∨ (OR) between the metrics indicates that they are optional measures to be collected, where any of them would be enough to analyze that characteristic for the Law of Software Evolution. This turns the framework more flexible, explaining the fact that not all the metrics need to be collected, turning it adaptable to the software development process that is being used.

This metrics set includes process and product metrics that should be collected for each produced OO software artifact version throughout the software life cycle. This will allow the definition of a baseline in order to study how the characteristics influence on OO software decay.

Table 2.12 – Interpretation of Metrics in the Requirements Specification Phase by Characteristic and Law of Software Evolution

	Size		Periodicity		Complexity		Effort		Modularity		Reliability		Efficiency		Maintainability
Continuing Change			¬↑ Interval between Versions			∧	¬↓ # Requirements Handled ∨ ¬↓ # Use Cases Handled								
Increasing Complexity	↑ # Function Points ∨ ↑ # Use Case Points			∨	↑ # Requirements ∨ ↑ # Use Cases	∨	↑ # Requirements Handled ∨ ↑ # Use Cases Handled	∨	↑ Coupling between Use Cases (# Extensions and Uses)					∨	↑ Spent Time in the Diagnostic of Defects ∨ ↑ Spent Time in the Removal of Defects
Self Regulation	¬↑ # Function Points ∨ ¬↑ # Use Case Points									∧	¬↑ # Detected Defects ∨ ¬↑ # Corrected Defects	∧	¬↓ # People ∨ ¬↓ Allocated Resources ∨ ¬↓ Spent Time ∨ ¬↓ Average Productivity of the Team		
Conservation of Organizational Stability							↔ # Requirements Handled ∨ ↔ # Use Cases Handled					∧	↔ # People ∧ ↔ Allocated Resources ∧ ↔ Spent Time ∧ ↔ Average Productivity of the Team		
Conservation of Familiarity	↔ # Function Points ∨ ↔ # Use Case Points			∧	↔ # Requirements ∨ ↔ # Use Cases	∧	↔ # Requirements Handled ∨ ↔ # Use Cases Handled								
Continuing Growth	↑ # Function Points ∨ ↑ # Use Case Points	∧	¬↑ Interval between Versions												
Declining Quality					↑ # Requirements	∨	↑ # Requirements	∨	↑ Coupling	∨	↑ # Detected			∨	↑ Spent Time in

	Size		Periodicity		Complexity		Effort		Modularity		Reliability		Efficiency		Maintainability
					∨ ↑ # Use Cases		Handled ∨ ↑ # Use Cases Handled		between Use Cases (# Extensions and Uses)		Defects ∨ ↑ # Corrected Defects				the Diagnostic of Defects ∨ ↑ Spent Time in the Removal of Defects
Feedback System	• # Function Points • # Use Case Points		• Interval between Versions		• # Requirements • # Use Cases		• # Requirements Handled • # Use Cases Handled		• Coupling between Use Cases (# Extensions and Uses)		• # Detected Defects • # Corrected Defects		• # People • Allocated Resources • Spent Time • Average Productivity of the Team		• Spent Time in the Diagnostic of Defects • Spent Time in the Removal of Defects

Table 2.13 – Interpretation of Metrics in the High Level Design Phase by Characteristic and Law of Software Evolution

	Size		Periodicity		Complexity		Effort		Modularity		Reliability		Efficiency		Maintainability
Continuing Change			¬↑ Interval between Versions			∧	¬↓ # Class Diagrams Handled ∨ ¬↓ # Sequence Diagrams Handled ∨ ¬↓ # State Diagrams Handled ∨ ¬↓ # Package Diagrams Handled ∨ ¬↓ # Activity Diagrams Handled								
Increasing Complexity	↑ # Classes ∨ ↑ # Methods per Class			∨	↑ # Class Diagrams ∨ ↑ # Sequence Diagrams ∨ ↑ # State Diagrams ∨ ↑ # Package Diagrams ∨ ↑ # Activity Diagrams ∨ ↑ Depth of Inheritance per Class ∨ ↑ # Children per Class	∨	↑ # Class Diagrams Handled ∨ ↑ # Sequence Diagrams Handled ∨ ↑ # State Diagrams Handled ∨ ↑ # Package Diagrams Handled ∨ ↑ # Activity Diagrams Handled	∨	↑ Coupling between Classes					∨	↑ Spent Time in the Diagnostic of Defects ∨ ↑ Spent Time in the Removal of Defects
Self Regulation	¬↑ # Classes ∨ ¬↑ # Methods per Class									∧	¬↑ # Detected Defects ∨ ¬↑ # Corrected Defects	∧	¬↓ # People ∨ ¬↓ Allocated Resources ∨ ¬↓ Spent Time		

	Size		Periodicity		Complexity		Effort		Modularity		Reliability		Efficiency		Maintainability
													∨ ¬↓ Average Productivity of the Team		
Conservation of Organizational Stability							↔ # Class Diagrams Handled ∨ ↔ # Sequence Diagrams Handled ∨ ↔ # State Diagrams Handled ∨ ↔ # Package Diagrams Handled ∨ ↔ # Activity Diagrams Handled					∧	↔ # People ∧ ↔ Allocated Resources ∧ ↔ Spent Time ∧ ↔ Average Productivity of the Team		
Conservation of Familiarity	↔ # Classes ∨ ↔ # Methods per Class			∧	↔ # Class Diagrams ∨ ↔ # Sequence Diagrams ∨ ↔ # State Diagrams ∨ ↔ # Package Diagrams ∨ ↔ # Activity Diagrams ∨ ↔ Depth of Inheritance per Class ∨ ↔ # Children per Class	∧	↔ # Class Diagrams Handled ∨ ↔ # Sequence Diagrams Handled ∨ ↔ # State Diagrams Handled ∨ ↔ # Package Diagrams Handled ∨ ↔ # Activity Diagrams Handled								
Continuing Growth	↑ # Classes ∨ ↑ # Methods per	∧	¬↑ Interval between Versions												

	Size		Periodicity		Complexity		Effort		Modularity		Reliability		Efficiency		Maintainability
	Class														
Declining Quality					↑ # Class Diagrams ∨ ↑ # Sequence Diagrams ∨ ↑ # State Diagrams ∨ ↑ # Package Diagrams ∨ ↑ # Activity Diagrams ∨ ↑ Depth of Inheritance per Class ∨ ↑ # Children per Class	∨	↑ # Class Diagrams Handled ∨ ↑ # Sequence Diagrams Handled ∨ ↑ # State Diagrams Handled ∨ ↑ # Package Diagrams Handled ∨ ↑ # Activity Diagrams Handled	∨	↑ Coupling between Classes	∨	↑ # Detected Defects ∨ ↑ # Corrected Defects			∨	↑ Spent Time in the Diagnostic of Defects ∨ ↑ Spent Time in the Removal of Defects
Feedback System	• # Classes • # Methods per Class		• Interval between Versions		• # Class Diagrams • # Sequence Diagrams • # State Diagrams • # Package Diagrams • # Activity Diagrams • Depth of Inheritance per Class • # Children per Class		• # Class Diagrams Handled • # Sequence Diagrams Handled • # State Diagrams Handled • # Package Diagrams Handled • # Activity Diagrams Handled		• Coupling between Classes		• # Detected Defects • # Corrected Defects		• # People • Allocated Resources • Spent Time • Average Productivity of the Team		• Spent Time in the Diagnostic of Defects • Spent Time in the Removal of Defects

Table 2.14 – Interpretation of Metrics in the Low Level Design Phase by Characteristic and Law of Software Evolution

	Size		Periodicity		Complexity		Effort		Modularity		Reliability		Efficiency		Maintainability
Continuing Change			¬↑ Interval between Versions			∧	¬↓ # Class Diagrams Handled ∨ ¬↓ # Sequence Diagrams Handled								
Increasing Complexity	↑ # Key Classes ∨ ↑ # Support Classes ∨ ↑ # Methods per Class ∨ ↑ # Subsystems			∨	↑ # Class Diagrams ∨ ↑ # Sequence Diagrams ∨ ↑ Depth of Inheritance per Class ∨ ↑ Coupling between Objects ∨ ↑ Response for a Class ∨ ↑ Lack of Cohesion in Methods ∨ ↑ # Children per Class	∨	↑ # Class Diagrams Handled ∨ ↑ # Sequence Diagrams Handled	∨	↓ Cohesion in Methods ∨ ↑ Coupling between Classes					∨	↑ Spent Time in the Diagnostic of Defects ∨ ↑ Spent Time in the Removal of Defects
Self Regulation	¬↑ # Key Classes ∨ ¬↑ # Support Classes ∨ ¬↑ # Methods per Class ∨ ¬↑ # Subsystems									∧	¬↑ # Detected Defects ∨ ¬↑ # Corrected Defects	∧	¬↓ # People ∨ ¬↓ Allocated Resources ∨ ¬↓ Spent Time ∨ ¬↓ Average Productivity of the Team		
Conservation of Organizational Stability							↔ # Class Diagrams Handled ∨ ↔ # Sequence Diagrams					∧	↔ # People ∧ ↔ Allocated Resources ∧ ↔ Spent Time		

	Size		Periodicity		Complexity		Effort		Modularity		Reliability		Efficiency		Maintainability
							Handled						∧ ↔ Average Productivity of the Team		
Conservation of Familiarity	↔ # Key Classes ∨ ↔ # Support Classes ∨ ↔ # Methods per Class ∨ ↔ # Subsystems			∧	↔ # Class Diagrams ∨ ↔ # Sequence Diagrams ∨ ↔ Depth of Inheritance per Class ∨ ↔ Coupling between Objects ∨ ↔ Response for a Class ∨ ↔ Lack of Cohesion in Methods ∨ ↔ # Children per Class	∧	↔ # Class Diagrams Handled ∨ ↔ # Sequence Diagrams Handled								
Continuing Growth	↑ # Key Classes ∨ ↑ # Support Classes ∨ ↑ # Methods per Class ∨ ↑ # Subsystems	∧	¬↑ Interval between Versions												
Declining Quality					↑ # Class Diagrams ∨ ↑ # Sequence Diagrams ∨ ↑ Depth of Inheritance per Class ∨ ↑ Coupling between Objects	∨	↑ # Class Diagrams Handled ∨ ↑ # Sequence Diagrams Handled	∨	↓ Cohesion in Methods ∨ ↑ Coupling between Classes	∨	↑ # Detected Defects ∨ ↑ # Corrected Defects			∨	↑ Spent Time in the Diagnostic of Defects ∨ ↑ Spent Time in the Removal of Defects

	Size		Periodicity		Complexity		Effort		Modularity		Reliability		Efficiency		Maintainability
					∨ ↑ Response for a Class ∨ ↑ Lack of Cohesion in Methods ∨ ↑ # Children per Class										
Feedback System	• # Key Classes • # Support Classes • # Methods per Class • # Subsystems		• Interval between Versions		• # Class Diagrams • # Sequence Diagrams • Depth of Inheritance per Class • Coupling between Objects • Response for a Class • Lack of Cohesion in Methods • # Children per Class		• # Class Diagrams Handled • # Sequence Diagrams Handled		• Cohesion in Methods • Coupling between Classes		• # Detected Defects • # Corrected Defects		• # People • Allocated Resources • Spent Time • Average Productivity of the Team		• Spent Time in the Diagnostic of Defects • Spent Time in the Removal of Defects

Table 2.15 – Interpretation of Metrics in the Code Phase by Characteristic and Law of Software Evolution

	Size		Periodicity		Complexity		Effort		Modularity		Reliability		Efficiency		Maintainability
Continuing Change			¬↑ Interval between Versions			∧	¬↓ # Lines of Source Code Handled								
Increasing Complexity	↑ # Lines of Source Code ∨ ↑ # Methods per Class			∨	↑ Depth of Inheritance per Class ∨ ↑ Coupling between Objects ∨ ↑ Response for a Class ∨ ↑ Lack of Cohesion in Methods ∨ ↑ # Children per Class ∨ ↑ Cyclomatic Complexity per Method	∨	↑ # Lines of Source Code Handled	∨	↓ Cohesion in Methods ∨ ↑ Coupling between Classes					∨	↑ Spent Time in the Diagnostic of Defects ∨ ↑ Spent Time in the Removal of Defects
Self Regulation	¬↑ # Lines of Source Code ∨ ¬↑ # Methods per Class									∧	¬↑ # Detected Defects ∨ ¬↑ # Corrected Defects ∨ ¬↓System Availability	∧	¬↓ # People ∨ ¬↓ Allocated Resources ∨ ¬↓ Spent Time ∨ ¬↓ Average Productivity of the Team		
Conservation of Organizational Stability							↔ # Lines of Source Code Handled					∧	↔ # People ∧ ↔ Allocated Resources ∧ ↔ Spent Time ∧ ↔ Average Productivity of the Team		
Conservation of	↔ # Lines of			∧	↔ Depth of	∧	↔ # Lines of								

	Size		Periodicity		Complexity		Effort		Modularity		Reliability		Efficiency		Maintainability
Familiarity	Source Code ∨ ↔ # Methods per Class				Inheritance per Class ∨ ↔ Coupling between Objects ∨ ↔ Response for a Class ∨ ↔ Lack of Cohesion in Methods ∨ ↔ # Children per Class ∨ ↔ Cyclomatic Complexity per Method		Source Code Handled								
Continuing Growth	↑ # Lines of Source Code ∨ ↑ # Methods per Class	∧	¬↑ Interval between Versions												
Declining Quality					↑ Depth of Inheritance per Class ∨ ↑ Coupling between Objects ∨ ↑ Response for a Class ∨ ↑ Lack of Cohesion in Methods ∨ ↑ # Children per Class ∨ ↑ Cyclomatic Complexity per Method	∨	↑ # Lines of Source Code Handled	∨	↓ Cohesion in Methods ∨ ↑ Coupling between Classes	∨	↑ # Detected Defects ∨ ↑ # Corrected Defects ∨ ↓ System Availability			∨	↑ Spent Time in the Diagnostic of Defects ∨ ↑ Spent Time in the Removal of Defects
Feedback System	• # Lines of Source Code • # Methods per		• Interval between Versions		• Depth of Inheritance per Class		• # Lines of Source Code Handled		• Cohesion in Methods • Coupling		• # Detected Defects • # Corrected		• # People • Allocated Resources		• Spent Time in the Diagnostic of Defects

	Size		Periodicity		Complexity		Effort		Modularity		Reliability		Efficiency		Maintainability
	Class				• Coupling between Objects • Response for a Class • Lack of Cohesion in Methods • # Children per Class • Cyclomatic Complexity per Method				between Classes		Defects • System Availability		• Spent Time • Average Productivity of the Team		• Spent Time in the Removal of Defects

3. Final Considerations and Future Perspectives

This work described a conceptual framework based on the Laws of Software Evolution to support the definition of experimental studies regarding object oriented software decay. It is intended to be applied in different object-oriented software development processes phases.

As suggestion for future works, there follows:

- Considering other relevant techniques, as Reuse of Software and Development Based on Components (LEHMAN & RAMIL, 2000), Families of Products (RIVA & ROSSO, 2002) and Software Architectures;
- Modifying the development process to insert collection phases and analysis of the metrics;
- Refining the set of hypotheses, goals, questions and metrics;
- Elaborating studies to evaluate how the characteristics relate to each other, throughout the software development process;
- Planning and executing experimental studies for evaluation of the proposed hypotheses, according to the experimentation process defined in (AMARAL & TRAVASSOS, 2003), using simulation techniques based on systems dynamics models (BARROS et al., 2004);
- Considering features of software rejuvenation (redocumentation, restructuring, reverse engineering, reengineering) (PFLEEGER, 2001);
- Proposing and building an environment that offers tool support for software evolution experimental studies.

References

AMARAL, E.A.G.; TRAVASSOS, G.H. 2003. A Package Model for Software Engineering Experiments. IEEE International Symposium on Empirical Software Engineering.

ARAUJO, M.A.; TRAVASSOS, G.H. 2004. Towards a Framework for Software evolution Experimental Studies, PESC Technical Report ES-641/04, COPPE/UFRJ. (available at http://www.cos.ufrj.br)

BARROS et al., 2004. Supporting Risks in Software Project Management". Journal Of Systems And Software.

BASILI & WEISS, 1984. A methodology for collecting valid software engineering data. IEEE Transactions on Software Engineering, Vol. SE-10, No. 6, 1984.

BASILI et al., 1996. Understanding and Predicting the Process of Software Maintenance Releases. Proc. 18th Int´l Conf. Software Eng., 1996.

BAUER & PIZKA, 2003. The Contribution of Free Software to Software Evolution. Sixth International Workshop on Principles of Software Evolution (IWPSE'03), IEEE, 2003.

BARRY et al, 1999. An Empirical Analysis of Software Evolution Profiles and Outcomes. Proceeding of the 20th international conference on Information Systems. ACM, 1999.

BELADY & LEHMAN, 1976. A Model of Large Program Development. IBM Systems J., vol. 15, no. 1, 1976.

BENDIFALLAH & SCACCHI, 1990. Understanding Software Maintenance Work. IEEE Trans. Software Engineering, 1990.

CAPILUPPI, 2003. Models for the Evolution of OS Projects. Proceedings of the International Conference on Software Maintenance. IEEE, 2003.

CAPILUPPI et al, 2003. Quantitative Models for Open Source Projects: A Proposal. 2003.

CAPILUPPI et al, 2003. Characteristics of Open Source Projects. Proceedings of the Seventh European Conference on Software Maintenance and Reengineering (CSMR´03). IEEE, 2003.

CARVER, 2003. The Impact of Background and Experience on Software Inspections. PhD Thesis, Faculty of the Graduate School of the University of Maryland, College Park. 2003.

CHIDAMBER & KEMERER, 1994. A Metrics Suite for Object Oriented Design. IEEE Transactions on Software Engineering, vol. 20, No. 6, 1994.

COOK & ROESCH, 1994. Real-Time Software Metrics. J. Systems and Software, vol. 24, no. 3, 1994.

CUSUMANO & YOFFIE, 1999. Software Development on Internet Time. Computer, October 1999.

EICK et al, 1999. Does Code Decay? Assessing the Evidence from Change Management Data. IEEE Computer, 1999.

GALL et al., 1997. Software Evolution Observations Based on Product Release History. Proc. 1997 Intern. Conf. Software Maintenance (ICSM´97), 1997.

GEFEN & SCHNEBERGER, 1996. The Non-Homogeneous Maintenance Periods: A Case Study of Software Modifications. Proc. Conf. Software Maintenance, IEEE, 1996.

GODFREY & TU, 2000. Evolution in Open Source Software: A Case Study. IEEE, 2000.

GODFREY & TU, 2001. Growth, Evolution, and Structural Change in Open Source Software. ACM, 2001.

GREENWOOD, et al, 1998. An Empirical Study of the Evolution of a Software System. Thirteenth IEEE Conference on Automated Software Engineering, 1998.

ISO 9126-1, 1997. International Standard. Information Technology – Software Quality Characteristics and Metrics – Part 1: Quality Characteristics and Sub-Characteristics, 1997.

KEMERER & SLAUGHTER, 1999, An Empirical Approach to Studying Software Evolution. IEEE, 1999.

KEMERER & SLAUGHTER, 1997. Determinants of Software Maintenance Profiles: An Empirical Investigation. Journal of Software Maintenance: Research and Practice, v.9 n.4, p.235-251, July-Aug. 1997.

KITCHENHAM, 1982. System evolution dynamics of VME/B. ICL Tech. J., 42-57, 1982.

LEHMAN, 1980. Programs, Life Cycle and the Laws of Software Evolution, Proc. IEEE, IEEE,1980.

LEHMAN & RAMIL, 2000. Software Evolution in the age of component-based software engineering. IEEE Software, 2000.

LEHMAN & RAMIL, 2001. Towards a Theory of Software Evolution – And its Practical Impact. IEEE, 2001.

LEHMAN & RAMIL, 2001b. Rules and Tools for Software Evolution Planning and Management. Annals of Software Engineering November 2001, vol. 11, no. 1, pp. 15-44(30), 2001.

LEHMAN & RAMIL, 2002. An Overview of Some Lessons Learnt in FEAST. WESS'02 Eighth IEEE Workshop on Empirical Studies of Software Maintenance, Canada, 2002.

LEHMAN & RAMIL, 2003. "Software Evolution – Background, Theory, Practice". Information Processing Letters, vol. 88, pp. 33 – 44, 2003.

LEHMAN, 1998. Software's Future: Managing Evolution. IEEE Software, 1998.

LEHMAN et al, 1998. Implications of Evolution Metrics on Software Maintenance. IEEE, 1998.

LEHMAN et al, 1997. Metrics and Laws of Software Evolution – The Nineties View. IEEE, 1997.

LORENZ & KIDD, 1994. Object-Oriented Software Metrics. Prentice-Hall, 1994.

MUNSON & WERRIES, 1996. Measuring Software Evolution. IEEE, 1996.

PARNAS, 1994. Software Aging. IEEE, 1994.

PERRY, 1994. Dimensions of Software Evolution. Proceedings of the International Conference on Software Maintenance. IEEE. 1994.

PERRY et al., 2001. Parallel Changes in Large-Scale Software Development: An Observational Case Study. ACM Trans. Software Engineering and Methodology, 2001.

PFLEEGER, 1998. The Nature of System Change. IEEE Software, 1998.
PFLEEGER, 2001. Software Engineering: Theory and Practice, 2th ed., Prentice Hall, 2001.
PRESSMAN, 2001. Software Engineering: A Practitioner's Approach, 5th ed., Mc-Graw Hill, 2001.
RAMIL, 2002. Laws of Software Evolution and their Empirical Support. Proceedings of the International Conference on Software Maintenance (ICSM´02), IEEE, 2002.
RIVA & ROSSO, 2002. Experiences with Software Product Family Evolution. Proceedings of the Sixth International Workshop on Principles of Software Evolution (IWPSE´03), IEEE, 2002.
ROSS, 1977. Structured Analysis for Requirements Definition. IEEE Transactions on Software Engineering, v. 3, n. 1, p. 6-15, jan, 1997.
SCACCHI, 2003. Understanding Open Source Software Evolution: Applying, Breaking, and Rethinking the Laws of Software Evolution. http://www.ics.uci.edu/~wscacchi/Papers/New/Understanding-OSS-Evolution.pdf, 2003.
SMITH & RAMIL, 2002. Qualitative Simulation of Software Evolution Process. WESS´02 Eighth Workshop on Empirical Studies of Software Maintenance, 2002.
SOLIGEN & BERGHOUT, 1999. The Goal/Question/Metric Method: a practical guide for quality improvement of software development. McGraw-Hill Publishing Company, 1999.
TAMAI & TORIMITSU, 1992. Software Lifetime and its Evolution Process over Generations. Proc. Conf. Software Maintenance, IEEE, 1992.
TRAVASSOS et al, 2001. Working with UML: A Software Design Process Based on Inspections for the Unified Modeling Language. Advances in Computers, 2001.
TRAVASSOS, 2003. Class notes of the discipline Special Topics in Software Engineering - Revisions, Inspection and Test of Object-Oriented Software. Systems Engineering and Computer Science Department – COPPE/UFRJ, 2003.
YUEN, 1985. An Empirical Approach to the Study of Errors in Large Software under Maintenance. Proc. Second Conf. Software Maintenance, IEEE, 1985.
YUEN, 1987. A Statistical Rationale for Evolution Dynamic Concepts. Proc. Conf. Software Maintenance, IEEE,1987.
YUEN, 1988. On Analyzing Maintenance Process Data at the Global and Detailed Levels: A Case Study. Proc. Fourth Conf. Software Maintenance, IEEE, 1988.